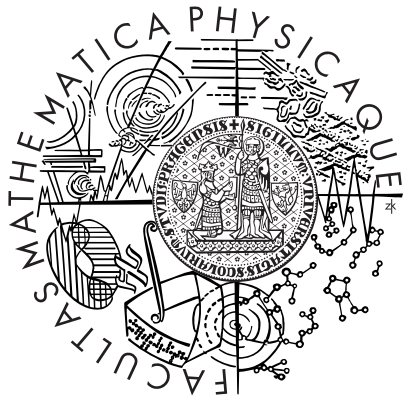
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Ema Krejčová

Algebraic Proof Systems

Department of Algebra

Supervisor: Prof. RNDr. Jan Krajíček, DrSc.
Study Program: Mathematics

2009

I would like to thank to my supervisor Jan Krajíček for his invaluable help, his patience and overall support.

I declare that I have written this master thesis on my own and listed all used sources. I agree with lending of the thesis.

Prague, 15th April 2009 Ema Krejčová

Contents

1 Essentials **6**
1.1 Languages 6
1.2 Turing machines 6
1.3 Algorithms 7
1.4 Decision problems 8

2 Computational complexity **10**
2.1 Complexity classes 10
2.2 Completeness 13

3 Proof complexity **15**
3.1 Proof systems 15
3.2 Algebraic proof systems 16

4 Compressed word problem **18**
4.1 Finitely presented monoids 18
4.2 Compression 20
4.3 Complexity of the problem 21

5 The CWC system **24**
5.1 Basic description 24
5.2 Extended CWC 29

6 Hard instances for CWC and E-CWC **30**

Název práce: Algebraické důkazové systémy
Autor: Ema Krejčová
Katedra: Katedra algebry
Vedoucí diplomové práce: Prof. RNDr. Jan Krajíček, DrSc.
e-mail vedoucího: krajicek@math.cas.cz

Abstrakt: Cílem této práce je navrhnout algebraický důkazový systém, který využívá poznatků z jiné oblasti algebry, než s jakou se v důkazové složitosti pracovalo dosud. Zde pracujeme s tzv. monoidy s konečnou prezentací, jak je definuje M. Lohrey [6]. Prvky monoidu (slova) kódujeme jako obvody, čímž je můžeme výrazně zkrátit, a definujeme důkazový systém CWC pro dokazování kongruencí takto kódovanych slov, zejména se soustředíme na problém, zda je určité slovo kongruentní s prázdným slovem. Tento problém je v určitém typu monoidů **coNP**-úplný, a to nám umožňuje vztáhnout tento systém i na základní **coNP**-úplný problém, kterým se důkazová složitost zabývá, totiž problém tautologií.

Klíčová slova: důkazová složitost, algebraické důkazové systémy, monoidy s konečnou prezentací, problém rovnosti slov

Title: Algebraic Proof Systems
Author: Ema Krejčová
Department: Department of Algebra
Supervisor: Prof. RNDr. Jan Krajíček, DrSc.
Supervisor's e-mail address: krajicek@math.cas.cz

Abstract: The aim of this thesis is to design a proof system that uses algebraic principles which have not yet been employed in proof complexity. Here, we work with finitely presented monoids, as they are described in M.Lohrey [6]. We use the encoding of monoid elements (words) as circuits (compressed words), which allows for a significant shortening, and we define a proof system, called CWC, for proving congruences of such compressed words, in particular, we concentrate on the problem whether a word is congruent to the empty word. In a specific type of monoids, this problem is **coNP**-complete, and therefore our system is related to the basic **coNP**-complete problem of proof complexity — the problem of tautologies.

Keywords: proof complexity, algebraic proof systems, finitely presented monoids, word problem

Introduction

The most important open question in complexity theory is whether **NP** is closed under complementation. Apart from the purely computational approach to the problem, there is another approach using formal proof systems. As Cook and Reckhow [2] proved, the $\mathbf{NP} =^? \mathbf{coNP}$ problem is equivalent to the question whether there is a **coNP**-complete language with a polynomially bounded proof system. Finding an answer to this question is the main topic in proof complexity.

There are basically two groups of proof systems. The first group consists of "traditional" proof systems which operate with formulas of propositional logic using a small set of inference rules. There are results showing superpolynomial lower bounds for some specific proof systems of this kind, but the means of propositional logic in these cases are limited. The second group consists of proof systems based on some algebraic or geometric principles. Algebraic proof systems extend significantly the possibilities of proof complexity, as they provide new methods for proving lower bounds and new heuristics for proof search. Algebraic systems described so far, such as Nullstellensatz system or Polynomial Calculus, work with polynomial identities over commutative rings. The aim of this thesis is to define an algebraic proof system operating in a different algebraic background, and thus to explore a new territory for proof complexity. As a starting point we take finitely presented monoids, described by M.Lohrey [6], which use a finite alphabet and a finite set of congruence rules, and we define a system for proving congruences of words, represented as circuits. This is called the compressed word problem. For a specific type of monoids, this problem is **coNP**-complete, and therefore provides a connection to the $\mathbf{NP} =^? \mathbf{coNP}$ problem.

In this thesis, we first describe the basic notions of complexity theory (Chapter 1), the issues of time and space complexity and the division of problems into complexity classes (Chapter 2) and the basic notions concerning proof systems, especially algebraic proof systems (Chapter 3). In Chapter 4, we describe the algebraic background for our proof system, which we call CWC (for Compressed Word Congruence) — finitely presented monoids, word problems and word compression. In Chapter 5, we define the CWC system itself and its extension E-CWC. Finally, in Chapter 6, we try to find some hard instances for these systems.

Chapter 1

Essentials

First, we introduce the basic notions which we need in order to be able to discuss the issues of complexity and provability.

This chapter mainly draws from Š. Holub's lecture notes [4] and C.H. Papadimitriou's book [7].

1.1 Languages

Definition 1.1 *An* alphabet Σ *is a set of symbols* $\{l_1, l_2, \ldots\}$*, these symbols are called* letters*. A* word *over an alphabet* Σ *is a sequence of letters from* Σ*.*

Although there can be infinite alphabets or words, here we will always consider finite alphabets and finite words. By the *length* of a word w, denoted $|w|$, we understand the number of letters in w. A word of length 0, the empty word, will be denoted ε.

Definition 1.2 *Given an alphabet* Σ*, a* language *over* Σ *is an arbitrary set of words over* Σ*.*

The "full language" of an alphabet Σ, i.e. the set of all finite words over Σ will be denoted Σ^*.

1.2 Turing machines

A Turing machine consists of a (potentially infinite) tape, a moving head, a memory and a program. The tape is divided into separate fields, each containing a letter of a given alphabet. At the beginning, the tape contains a finite word, the *input* of the machine. The head can read the content of a field, change it, and move to the next field to the right or to the left, or stay in place. The memory always takes one of a finite number of values, called *states*. The program directs the processing of the machine according to the current state and current letter read by the head.

In the mathematical sense, we are not interested in the "hardware" of the Turing machine, just in its program, the alphabet and the set of states:

Definition 1.3 *Let Σ be a finite alphabet with at least two letters, and let $\varnothing$ be a symbol for a blank field, $\varnothing \notin \Sigma$. Let M be a set of three elements $\{\rightarrow, -, \leftarrow\}$. Let Q be a finite set and q_i be one particular element in Q. The elements of Q are called* states*, q_i is called the* initial state*.*
A Turing program *P is a set of quintuples of the form (q_1, s_1, q_2, s_2, m) where $s_1, s_2 \in \Sigma \cup \{\varnothing\}, q_1, q_2 \in Q$ and $m \in M$. These quintuples are called* instructions*.*
A Turing machine *T is a quadruple (Σ, Q, q_i, P)*

In the "hardware picture" of a Turing machine, an instruction of the form (q_1, s_1, q_2, s_2, m) says: if the current state is q_1 and the head reads s_1, in other words if the current *configuration* is (q_1, s_1), then change the current state to q_2, erase the letter s_1, write s_2 instead and move the head according to m.

A Turing machine T together with a finite word w over the alphabet Σ define a *computation* of T. The word w is called the *input*. At the beginning, the input is written on the tape and all the other fields contain the blank symbol $\varnothing$. The computation of a Turing machine consists of individual *steps*, in each step one instruction is applied, according to the current configuration of the Turing machine. The initial configuration is (q_i, s_i), where q_i is the initial state and s_i is the first letter of the input.

The computation halts, if there is no instruction for the current configuration. The sequence of all non-blank symbols which are on the tape at the moment when the computation halts is called the *output*.

When we write $T(x) = y$ we mean "the Turing machine T halts on input x and gives the output y". Sometimes we are not interested in the value of the output as long as there is an output, in this case we say that the Turing machine T *converges* on the input x and we denote it $T(x)\downarrow$. If T on x does not halt at all, we say that T *diverges* on x and denote it $T(x)\uparrow$.

Definition 1.4 *A Turing machine is* deterministic *(DTM), if for every configuration (q_1, s_1) there is at most one instruction of the form (q_1, s_1, q_2, s_2, m).*
A Turing machine is nondeterministic *(NTM), if there is a pair of instructions $(q_1, s_1, q_2, s_2, m_2)$, $(q_1, s_1, q_3, s_3, m_3)$ such that $(q_2, s_2, m_2) \neq (q_3, s_3, m_3)$.*

The computation of a DTM on a given input is unique. A NTM can perform different computations with different results on the same input.

1.3 Algorithms

When solving a problem, we are usually not interested in the solution for one particular case, we are looking for an *algorithm* which will give an answer for every instance of the problem.

A Turing machine is an attempt to formalize our intuitive notion of an algorithm. There is no precise definition of an algorithm, but informally we can say it is a method of computation, or more generally a method of finding a solution to a problem. However, not every method will comply with our notion of an algorithm, we expect it to have certain properties:

- to have explicit rules of operation which can be applied mechanically
- to have a rule for every situation which can occur during the computation
- to work universally for each possible input

The claim that the Turing machine model can describe every computational method which we would consider an algorithm in the above sense is known as the *Church-Turing thesis*.

As there is no formal definition of an algorithm, Church-Turing thesis cannot be proved. However, it is widely accepted, and all other models formalizing algorithms were proved to be equivalent to the Turing machine (or weaker).

Therefore, the converse of Church-Turing thesis is often used as a definition of an algorithm, in the sense that "what is computable by a Turing machine, is an algorithm".

In the following, we will use the terms algorithm and Turing machine as equivalent.

1.4 Decision problems

We can sort problems according to the type of the expected answer. A *decision problem* is a question with a yes/no answer, depending on some input value. A decision problem can be also described as a function with the domain consisting of all possible instances of the problem and the range $\{0, 1\}$. A problem which requires a more complex answer is called a *function problem*. Usually, the set of possible answers to a function problem is infinite.

In the following, we will concentrate on decision problems, as they are much easier to formalize — they are closely related to languages, as defined in 1.1: Each instance of a problem P can be encoded as a word over some alphabet Σ. We define a language $L_P \subset \Sigma^*$ consisting of all words which encode instances with the answer 1. The decision problem P is now equivalent to the characteristic function of the language L_P (the answer is 1 if and only if the corresponding word belongs to L_P). Therefore, decision problems can be considered equivalent to languages.

To describe the behaviour of deterministic and nondeterministic Turing machines on languages, we will use the following terms:

Definition 1.5 *Let T be a DTM. We say that T* decides *a language $L \subset \Sigma^*$ if for every $x \in \Sigma^*$:*

- *if $x \in L$ then $T(x) = 1$*
- *if $x \notin L$ then $T(x) = 0$*

We say that T accepts *$L \subset \Sigma^*$ if for every $x \in \Sigma^*$:*
$x \in L$ iff $T(x)\downarrow$

Accepting is significantly different from deciding. The notion of accepting is asymmetric in the following sense — a T that accepts L diverges on inputs not belonging to L. Therefore, if T halts on input x, we know that $x \in L$, but while T is still computing, we know nothing — the computation might halt after a few more steps, or it might go on forever.

Definition 1.6 *Let T be a NTM. We say that T* accepts *a language $L \subset \Sigma^*$ if for every $x \in \Sigma^*$:*

- *$x \in L$ iff there exists at least one computation of T on x such that $T(x) = 1$*

The asymmetry of acceptance is similar as with DTMs. With a NTM, when a computation on input x halts with the output 1, we know that $x \in L$. But if it halts with output 0, or does not halt yet, we do not know whether there is another computation on the same input which would give the output 1. In fact, a NTM can be always simulated by a DTM — the corresponding DTM makes a breadth-first search of all the possible computations of the NTM (it runs the first step of each computation, then the second, and so on).

Chapter 2

Computational complexity

2.1 Complexity classes

An important question always is whether there exists a deciding algorithm for a particular problem. There are problems, such as the *halting problem* (see e.g. [7], p.58), which cannot be solved by a Turing machine. But when we already have an algorithm solving a particular problem, another important question is how "effective" the algorithm is — we speak about the *complexity* of the algorithm.

The complexity is always measured as a function of the length of the input word. Here, we will concentrate on the time complexity, but we mention also the space complexity for the sake of completeness of the presentation.

Definition 2.1 *Let M be a Turing machine. Let $t'_M(x)$ denote the minimum number of steps that M performs before halting on the input x. The* time complexity *of M is a function $t_M : \mathbb{N} \to \mathbb{N} \cup \{\infty\}$,*

$$t_M(n) = \max_{|x| \leq n} \{t'_M(x)\}$$

Let $s'_M(x)$ denote the minimum number of fields visited by the head during the computation of M on the input x. The space complexity *of M is a function $s_M : \mathbb{N} \to \mathbb{N} \cup \{\infty\}$,*

$$s_M(n) = \max_{|x| \leq n} \{s'_M(x)\}$$

We consider the minimum numbers in $t'_M(x)$ and $s'_M(x)$ because M is generally nondeterministic. We define $t_M(n)$ and $s_M(n)$ with maximums taken over all words of length less or equal to n to ensure that all the complexity functions are nondecreasing.

The definition of time and space complexity works for function problems as well as for decision problems, but here we are mainly interested in the decision problems, or languages.

We divide languages according to their complexity into complexity classes:

Definition 2.2 *Let $f : \mathbb{N} \to \mathbb{N}$ be a nondecreasing function. We say that a language L belongs to the class* $\mathbf{TIME}(f)$*, if there exists a DTM T which decides L and for the time complexity of T it holds that $t_T \leq f$.*
We say that L belongs to $\mathbf{SPACE}(f)$*, if there is a DTM T which decides L and for the space complexity of T it holds that $s_T \leq f$.*
Analogously, if we replace DTMs by NTMs and deciding by accepting in the preceding definitions, we obtain nondeterministic classes $\mathbf{NTIME}(f)$ *and* $\mathbf{NSPACE}(f)$*.*

Instead of a single function f, we can use a family of functions F. Then, $\mathbf{TIME}(F)$ denotes the union of all classes $\mathbf{TIME}(f)$ for all $f \in F$.

Generally, f may be an arbitrary function and it may be quite complicated. However, it can be approximated by a simplier function and we will show that this approximation does not change the complexity. We will use the following notation:

Definition 2.3 *Let f, g be functions $\mathbb{N} \to \mathbb{N}$. We say that $f \in \mathcal{O}(g)$ if there is a constant $c > 0$ such that for all $n \in \mathbb{N}$*

$$f(n) \leq c \cdot g(n)$$

Proposition 2.4 (Linear Speedup) *For every function $f : \mathbb{N} \to \mathbb{N}$,*

$$\mathbf{TIME}(f(n)) = \mathbf{TIME}(\mathcal{O}(f(n)))$$

$$\mathbf{SPACE}(f(n)) = \mathbf{SPACE}(\mathcal{O}(f(n)))$$

Proof: We will show that any language L which belongs to $\mathbf{TIME}(f(n))$ and $\mathbf{SPACE}(g(n))$ also belongs to $\mathbf{TIME}(c \cdot f(n))$ and $\mathbf{SPACE}(c \cdot g(n))$ for arbitrary $c > 0$.

Let M be a Turing machine which decides L in $f(n)$ steps and using $g(n)$ fields of the tape. We shall construct a Turing machine M' which decides L in $c \cdot f(n)$ steps and within $c \cdot g(n)$ fields.

Let k be a constant, which will be specified later. If the alphabet of M is Σ, then the alphabet of M' is Σ' which consists of all k-tuples of symbols in Σ. The content of the tape of M' will be the same as that of M throughout the computation, it will be just grouped into k-tuples. Therefore the space used by M' will be $\frac{g(n)}{k}$.

M' simulates every k steps of the computation of M by one cycle: The head of M' reads a k-tuple which contains the symbol read by the head of M. During the k steps of M, only this particular k-tuple and one of its two neighbors may be changed. So, M' has to read these three k-tuples (it can "remember" the content within its states) and change their content according to M. This can be done in 6 steps (read the symbol, go left - read the symbol,

go right - once more go right - read the symbol, go left - change the symbol, go to the affected neighbor - change the symbol, go back or stay, depending on where the head of M is). Therefore, the time needed by M' is at most $\frac{6}{k}f(n)$.

Now, if we put $k = \lceil \frac{6}{c} \rceil$ we get the intended result. □

When judging complexity, Proposition 2.4 allows us to take into account only the leading term of the complexity function and to ignore the coefficients (for example it is legitimate to use $\mathbf{TIME}(n^3)$ instead of $\mathbf{TIME}(4n^3 + 7n)$).

Some of the important complexity classes are:

Definition 2.5

$$\begin{aligned} \mathbf{P} &= \bigcup_{k \in \mathbb{N}} \mathbf{TIME}(n^k) \\ \mathbf{NP} &= \bigcup_{k \in \mathbb{N}} \mathbf{NTIME}(n^k) \\ \mathbf{PSPACE} &= \bigcup_{k \in \mathbb{N}} \mathbf{SPACE}(n^k) \\ \mathbf{EXP} &= \bigcup_{k \in \mathbb{N}} \mathbf{TIME}(2^{n^k}) \end{aligned}$$

Identifying decision problems with languages, i.e. subsets of Σ^* for some alphabet Σ, we can define a *complementary problem* to a problem L as $\Sigma^* \setminus L$. We denote this problem *co*L. Similarly, if $\mathbf{C}$ is a class of problems, then $\mathbf{coC}$ denotes a *complementary class* to $\mathbf{C}$ which contains just complementary problems to those in $\mathbf{C}$.

There are many open questions concerning relations between complexity classes.

It is known (and proved e.g. in [7], pp.143–148) that

$$\mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{PSPACE} \subseteq \mathbf{EXP}$$

and also that

$$\mathbf{P} \subsetneq \mathbf{EXP}$$

But the exact position of $\mathbf{NP}$ is not clear. Above all, the relation between $\mathbf{P}$ and $\mathbf{NP}$ is of a great interest. It is conjectured that

$$\mathbf{P} \neq \mathbf{NP}$$

but so far, there is no proof. The conjecture is that all the above inclusions are proper, but we only know for sure that one of them is.

Another important unresolved question is the relation between **NP** and **coNP**. We know that

$$\mathbf{P} = \mathbf{coP}$$

because if there is a deterministic Turing machine M which decides a language $L \in \mathbf{P}$, we can easily obtain a machine M' which decides coL just by flipping the output values of M. However, with nonedeterministic machines, we do not have this easy way because of the possible diverging computations.

If it could be proved that $\mathbf{P} = \mathbf{NP}$ then we would get the result that $\mathbf{NP} = \mathbf{coNP}$, but the assumption is considered unlikely. It is conjectured that

$$\mathbf{NP} \neq \mathbf{coNP}$$

but again, there is no proof so far.

2.2 Completeness

Problems within a complexity class are not completely independent. If we have an algorithm to solve a certain problem, we can often modify it for a different problem — we say that one problem reduces to another. It is an efficient way to compare complexity of problems, on the assumption that the reduction is not too "expensive", i.e. that the reduction itself does not increase the complexity too much.

Definition 2.6 *Let Σ_1, Σ_2 be alphabets. We say that a function $f : \Sigma_1^* \to \Sigma_2^*$ is* polynomially computable *if there exists a deterministic Turing machine T such that for every $x \in \Sigma^*$: $T(x) \downarrow$ and $T(x) = f(x)$, and the time complexity of T is bounded by a polynomial, i.e. $t_T(n) \leq n^c$ for some constant c.*

Definition 2.7 *Let L_1, L_2 be languages over alphabets Σ_1, Σ_2 respectively. We say that L_1 is* polynomially reducible *to L_2 if there exists a polynomially computable function $f : \Sigma_1^* \to \Sigma_2^*$ such that*

$$\forall x \in \Sigma_1^* : x \in L_1 \text{ iff } f(x) \in L_2$$

We say that a complexity class **C** *is* closed under polynomial reduction *if the following holds: if L_1 is polynomially reducible to L_2 and $L_2 \in \mathbf{C}$, then also $L_1 \in \mathbf{C}$.*

Proposition 2.8 **P** *and* **NP** *are closed under polynomial reduction.*

Proof: Suppose L_2 is a language in **P** and L_1 is polynomially reducible to L_2. Let f be the polynomially computable function that ensures the reducibility, M_f be the DTM that computes f and M_2 be the DTM that decides L_2. Time complexity of both M_f and M_2 is bounded by a polynomial, say:

$$t_{M_f}(n) \leq n^c$$

$$t_{M_2}(n) \leq n^d$$

We construct a DTM M_1 which decides L_1 as a composition $M_1 = M_2 \circ M_f$ and show that time complexity of M_1 is also bounded by a polynomial:

$$t_{M_1}(n) = t_{M_2}(t_{M_f}(n)) \leq t_{M_2}(n^c) \leq (n^c)^d = n^{c \cdot d}$$

If L_2 is in **NP**, we take M_2 as a NTM, and by an analogous construction we obtain a polynomially bounded NTM M_1 for L_1. □

We can use reducibility to pick "the most difficult" member of a given class.

Definition 2.9 *Let* **C** *be a complexity class which is closed under polynomial reduction. We say that a language L is* **C**-hard *if every $L' \in \mathbf{C}$ is polynomially reducible to L. We say that L is* **C**-complete *if L is* **C**-*hard and moreover $L \in \mathbf{C}$.*

A complete problem is a valuable representative of a class — it justifies the existence of its particular class: if we can show that a **C**-complete language L belongs to $\mathbf{C}' \subseteq \mathbf{C}$, then **C** and $\mathbf{C}'$ coincide.

The notion of completeness is mainly discussed with respect to the class **NP**, since many natural problems were proved to be **NP**-complete.

One of the basic **NP**-complete problems is Boolean satisfiability (SAT) — a decision problem whose input is a Boolean formula and the question is whether there exists an assignment of truth values to the variables in the formula such that the entire formula is true.

The statement that SAT is **NP**-complete was independently proved by Steven Cook and Leonid Levin in the 1970's and it is known as the Cook's Theorem or Cook-Levin Theorem (the proof can be found e.g. in [3], pp.38–44).

Chapter 3

Proof complexity

When we examine computational complexity of a problem, we are interested in the algorithm which finds the solution to the problem. In proof complexity, we are not interested in the algorithm, but we want to provide a token that witnesses the answer, a "proof". And we would like these proofs to be as short as possible.

With some problems, we can easily provide such a proof. For example, if we ask whether an integer n is a composite number, any proper factor of n will serve as a proof in the affirmative case. Or, if we take a Boolean formula f, an assignment of truth values which makes f true can be a proof of satisfiability of f. Generally, if we want to prove that certain object exists, the object itself can be the "proof". On the other hand, if we want to prove that an object with a certain property does not exist, it is more complicated — basically we have to examine all possible objects and show that none has the property. In this case, what should the "proof" look like?

First, we formalize the notion of a proof (as it is defined in [2] and [5]).

3.1 Proof systems

Definition 3.1 *Let $L \subset \Sigma^*$ be a language. A* proof system *for L is a binary relation $P(x, y) \subset \Sigma^* \times \Sigma^*$ satisfying the following conditions:*

- *Completeness: if $x \in L$ then $\exists y : P(x, y)$*
- *Soundness: if $\exists y : P(x, y)$ then $x \in L$*
- *p-verifiability: $P(x, y)$ is decidable by a deterministic Turing machine in polynomial time.*

If $P(x, y)$ holds, we say that y is a *P-proof* of x.

Definition 3.2 *A proof system P is* p-bounded *if there exists a polynomial $q(n)$ such that for every $x \in L$ there is y such that $P(x, y)$ and $|y| \leq q(|x|)$.*

In Definition 2.5 we have described the class **NP** in terms of computational complexity. But there is an equivalent description of **NP** using proof systems — Cook and Reckhow in [2] prove the following statement:

Proposition 3.3 *A language L is in* **NP** *iff $L = \emptyset$ or L has a p-bounded proof system.*

This allows for a new approach to the question whether **P** differs from **NP** and **NP** from **coNP**. If we can prove for some language L in **coNP** that there does not exist a p-bounded proof system for L, we would get the result that **NP** $\neq$ **coNP** (and hence **P** $\neq$ **NP**, too).

A suitable representative of the class **coNP** is the language TAUT, which consists of all propositional tautologies. TAUT is **coNP**-complete, since it is a complementary problem to SAT — proving that a formula is not satisfiable is equivalent to proving that its negation is a tautology. Proof systems for TAUT are called *propositional proof systems*.

The following statement (also proved in [2]) reformulates the question of **NP** $\neq$ **coNP**:

Proposition 3.4 *The class* **NP** *is closed under complementation iff there exists a p-bounded proof system for TAUT.*

The main goal of proof complexity is to show that such a proof system does not exist. Or, to find one...

3.2 Algebraic proof systems

Traditional propositional proof systems, usually called *Frege systems*, work directly on formulas, using a finite number of axioms and inference rules for derivation. However, the methods of proving in logic can be extended by using algebraic principles. An *algebraic proof system* in a narrower sense is a system which operates with polynomials over some commutative ring and attempts to prove that a set of polynomial equations has no solution.

In a more general sense, an algebraic proof system is any system which uses algebraic structures and operations.

Algebraic proof systems are closely connected to Frege systems — it is possible to translate formulas into polynomials (for the translation see [1]) and instead of proving that a set of formulas is unsatisfiable prove that a set of equations is unsolvable.

A prototypical example of such an algebraic system is the system based on Hilbert's Nullstellensatz:

Theorem 3.5 (Weak Nullstellensatz) *Let K be an algebraically closed field and $S = K[x_1, \ldots, x_n]$ be a ring of polynomials over K in n variables. For every proper ideal I in S, the set of zeros*

$$V(I) = \{(a_1, \ldots, a_n) \in K^n \text{ such that } f(a_1, \ldots, a_n) = 0 \text{ for every } f \in I\}$$

is nonempty.

The Nullstellensatz system (described more precisely in [5] and [1]) is a proof system for the language L, which consists of sets of polynomial equations over some finite field F which have no 0-1 solution. Let $\{f_i = 0\}$ for $i \in \{1, \ldots, k\}$ be a set of polynomial equations (in n variables). A Nullstellensatz proof of its unsolvability is a list of polynomials g_i for $i \in \{1, \ldots, k\}$ and h_j for $j \in \{1, \ldots, n\}$ such that the identity

$$\sum_{i=1}^{k} g_i \cdot f_i + \sum_{j=1}^{n} h_j \cdot (x_j^2 - x_j) = 1$$

holds in the ring of polynomials over F.

We work with a finite F to ensure simple encoding of the coefficients in some finite alphabet. Hilbert's theorem requires an algebraically closed field, which F is not, but thanks to the presence of the polynomials $(x_j^2 - x_j)$ which restrict the solutions only to 0 and 1, the condition is not necessary (for a proof see [1]).

The Nullstellensatz system is a proper proof system in the sense of Definition 3.1. It is sound — if there are such g_i that the equality holds, then there cannot exist a solution to all f_i. If there was an element $a = (a_1, \ldots, a_n) \in F^n$ such that $f_i(a) = 0$ for every i, then also the sum would be equal to 0. The system is complete — if the set of f_i is unsolvable, its set of zeros is empty, therefore by 3.5 the ideal generated by f_i is the whole field and the corresponding g_i exist. The equality is p-verifiable, as long as the polynomials are well represented (e.g. as sums of monomials).

Chapter 4

Compressed word problem

Our aim in this thesis is to define a new algebraic proof system that operates in a different algebraic background than earlier systems working with polynomial equations. We are going to construct a propositional proof system based on a specific algebraic structure — finitely presented monoids, as they are described in M.Lohrey's paper [6].

An alphabet A together with the associative operation of concatenation forms a semigroup G, whose elements are all words in A^*. We will denote the concatenation of letters a, b simply by ab. Similarly, uv will denote the concatenation of words u, v. The empty word ε is the unit element in G, so we can say that G is a monoid.

By introducing congruence rules on the words in G, we can create various monoids, and we can ask whether two words over A represent the same element of the monoid — this is called the *word problem*. Lohrey in [6] examines the properties of these monoids with respect to the congruence rules and also the complexity of the corresponding word problems. Moreover, he introduces a system of compression of the words and proves that for a specific type of monoids, the problem whether a compressed word is congruent to the empty word, is **coNP**-complete and that is exactly what we need to construct our proof system — if we have a proof system for a **coNP**-complete language L, we also have implicitly a proof system for TAUT, because TAUT is then reducible to L.

4.1 Finitely presented monoids

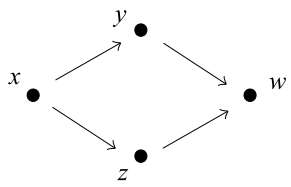
Definition 4.1 *Let A be a finite alphabet and R be a finite set of rules of the form $(x \to y)$, where $x, y \in A^*$. The pair (A, R) is called a* monoid presentation.
Let w_1, w_2 be words over A. We define the following relations:

- $w_1 \to_R w_2$ *if $w_1 = w_2$ or if there are words $u, v \in A^*$ and a rule $(x \to y) \in R$ such that $w_1 = uxv$ and $w_2 = uyv$ (we say that w_1 is* reducible *to w_2)*

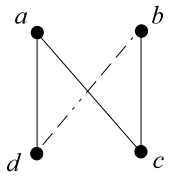
- $w_1 \leftrightarrow_R w_2$ *if* $w_1 \rightarrow_R w_2$ *or* $w_2 \rightarrow_R w_1$
- $w_1 \rightarrow_R^* w_2$ *if there are* $x_1, x_2, \ldots, x_n \in A^*$ *such that* $w_1 \rightarrow_R x_1 \rightarrow_R \ldots \rightarrow_R x_n \rightarrow_R w_2$
- $w_1 \leftrightarrow_R^* w_2$ *if there are* $x_1, x_2, \ldots, x_n \in A^*$ *such that* $w_1 \leftrightarrow_R x_1 \leftrightarrow_R \ldots \leftrightarrow_R x_n \leftrightarrow_R w_2$

Definition 4.2 *Let* (A, R) *be a monoid presentation.* Word problem *for* (A, R) *is the following decision problem: given two words* $w_1, w_2 \in A^*$*, is it true that* $w_1 \leftrightarrow_R^* w_2$*?*

Definition 4.3 *We say that a presentation* (A, R) *is* terminating*, if there does not exist an infinite chain* $w_1 \rightarrow w_2 \rightarrow w_3 \rightarrow \ldots$ *with* $w_i \neq w_{i+1}$ *for all* $i \in \mathbb{N}$*. A presentation* (A, R) *is* confluent*, if for every* $x, y, z \in A^*$ *such that* $x \rightarrow_R^* y$ *and* $x \rightarrow_R^* z$ *exists* $w \in A^*$ *such that* $y \rightarrow_R^* w$ *and* $z \rightarrow_R^* w$*.*



We say that a presentation (A, R) *is* 2-homogeneous*, if every rule in* R *is of the form* $(ab \rightarrow \varepsilon)$ *where* a, b *are letters of* A*. We say that a 2-homogeneous presentation is* N-free *if there are no* $a, b, c, d \in A$ *such that* R *contains* $(ac \rightarrow \varepsilon)$*,* $(ad \rightarrow \varepsilon)$ *and* $(bc \rightarrow \varepsilon)$*, but does not contain* $(bd \rightarrow \varepsilon)$*.*



Definition 4.4 *Let* (A, R) *be a monoid presentation. A word* $w_1 \in A^*$ *is called* irreducible *if there is no* $w_2 \neq w_1 \in A^*$ *such that* $w_1 \rightarrow_R^* w_2$*.*

If a presentation (A, R) is terminating and confluent, then for every $w \in A^*$ there is a unique irreducible word z such that $w \rightarrow_R^* z$. We say that z is a *normal form* of w, denoted $z = NF(w)$.

The relation $\leftrightarrow_R^*$ is a congruence relation on A^*, so we can split A^* into equivalence classes and define the following:

Definition 4.5 *A* finitely presented monoid $M(A, R)$ *is a quotient monoid* $A^* / \leftrightarrow_R^*$*.*

A monoid presentation is not unique, a monoid can have many different presentations, but the decidability and complexity of its word problem does not depend on the choice of presentation, therefore we can talk about the word problem of a monoid.

If the monoid has a terminating and confluent presentation, its word problem is decidable. In this case, each word has an irreducible normal form and in each equivalence class of $\leftrightarrow_R^*$, there is exactly one irreducible word, so all words in the same class have the same normal form. Therefore, to check whether $w_1 \leftrightarrow_R^* w_2$ it is enough to show that $NF(w_1) = NF(w_2)$. (For further reference see [6]).

4.2 Compression

Using an efficient way of encoding, we can reasonably shorten the space needed to write down a word. Expressing a word in a compressed way (as a circuit, in our case) can be significantly shorter than writing it explicitly as sequence of letters.

Definition 4.6 *Let A be a finite alphabet. A* circuit *C over A consists of n variables $x_1, x_2, \ldots, x_n$ and n instructions of the form $x_i \to w_i$ for $i \in \{1, 2, \ldots, n\}$, where w_i is one of the following:*

- *$w_i = l$, where l is a letter of A or $l = \varepsilon$*
- *$w_i = x_j x_k$, where $j, k < i$*

The number of variables in a circuit C will be called the *size* of C.
The set of instructions of C will be denoted $Ins(C)$.

Circuits are built from potentially infinitely many letters — apart form the alphabet A, which is finite, they use also letters for variables $x_1, x_2, x_3, \ldots$ To conform with the restriction to finite alphabets we think of x_n as being the symbol x followed by the binary encoding of n.

Definition 4.7 *Let C be circuit of size n. For $i \in \{1, 2, \ldots, n\}$, we define $val(C, x_i)$, the* value of C at x_i *recursively:*

- *if the i-th instruction is $x_i \to l$ where $l \in A$ or $l = \varepsilon$, then*

$$val(C, x_i) = l$$

- *if the i-th instruction is $x_i \to x_j x_k$, then*

$$val(C, x_i) = val(C, x_j) val(C, x_k)$$

(the concatenation of the respective values).

We define the value of C *as the value at the last variable, i.e.*

$$val(C) = val(C, x_n)$$

The value of a circuit is always a word over the alphabet A. In this way, each circuit represents a word, therefore we say that the circuit is a *compressed word* over A. We are going to use the terms circuit and compressed word as equivalent.

Example 4.8 *Let us examine a word defined by the Fibonacci sequence. In its circuit form, it looks like this:*

$x_1 \to b$
$x_2 \to a$
$x_i \to x_{i-1}x_{i-2} \quad$ *for* $i \geq 3$

With growing n, the space needed to write down the n-th Fibonacci word in compressed form grows only linearly — we need n instructions, each containing at most 4 symbols. If we want to write down the word explicitly, its size grows exponentially:
$ab - aba - abaab - abaababa - abaababaabaab - \ldots$

We can examine the word problem for a monoid also in relation with compression — the *compressed word problem* is the question whether two circuits represent the same word.

4.3 Complexity of the problem

Lohrey in [6] proves the following results about the time complexity of the compressed word problem:

Theorem 4.9 *Let M be a monoid with a confluent, 2-homogeneous, N-free presentation (A, R). Then the compressed word problem for M is in* **P**.

Theorem 4.10 *Let M be a monoid with a confluent, 2-homogeneous, non-N-free presentation (A, R). Then the following decision problem (a special case of the compressed word problem):*

- *For a given compressed word C over A, does $val(C) \leftrightarrow_R^* \varepsilon$ hold?*

is **coNP***-complete.*

We are not going to repeat Lohrey's proof in detail, we just mention the particular monoid we are going to use for the construction of our system, and we outline the proof that for this monoid, the problem is in **coNP**.

We fix a monoid $\mathcal{M} = M(A, R)$, where

$$\begin{aligned} A &= \{a, b, c, d\} \\ R &= \{(ac \to \varepsilon), (ad \to \varepsilon), (bc \to \varepsilon)\} \end{aligned}$$

(A, R) is a confluent, 2-homogeneous, non-N-free presentation.

We introduce some notation to refer to the decision problem from Theorem 4.10:

Definition 4.11 *Loh is the language which consists of all circuits C over the alphabet A such that $val(C) \to_R^* \varepsilon$.*

To prove that the problem whether a circuit C is in *Loh* is **coNP**-complete, we have to show that it is **coNP**-hard and that it belongs to **coNP**.

To show the latter, we proceed as follows:
We define a function $\rho : A^* \to \{\langle, \rangle\}^*$ which maps letters a, b to '$\langle$' and c, d to '$\rangle$', and for all words $x, y \in A^* : \rho(xy) = \rho(x)\rho(y)$.
For a compressed word C over A, we can compute a compressed word D over $\{\langle, \rangle\}^*$ such that $val(D) = \rho(val(C))$ in polynomial time — we just have to go through the circuit C once and replace all letters by '$\langle$' and '$\rangle$'.
Our monoid presentation (A, R) is thus mapped to

$$(B, S) = (\{\langle, \rangle\}, \{(\langle\rangle \to \varepsilon)\})$$

which is still confluent and 2-homogeneous and moreover it is N-free, so its compressed word problem is decidable in polynomial time, by Theorem 4.9.
We say that $val(D)$ is a *well-bracketed* word, if $val(D)$ is reducible to ε using the rule $(\langle\rangle \to \varepsilon)$. In a well-bracketed word, for every '$\langle$' there is a unique '$\rangle$' such that the reducing rule will be applied on the pair $\langle\rangle$ during the reduction of the word. If the '$\langle$' symbol is on the i-th position in $val(D)$, we denote the position of the corresponding '$\rangle$' by $match(i)$.

For a given i, we can compute $match(i)$ in polynomial time (this is proved in [6] as Lemma 5.6).

Given a compressed word C and a position i, we can compute the i-th symbol in $val(C)$ in polynomial time. We do not have to expand the whole word, we just have to go through the circuit twice. First bottom-up and compute the lengths of subsequent words, then top-down and look only into the appropriate subword of the word constructed.

All this put together gives us a **NP** algorithm which accepts compressed words C over A, such that $val(C) \not\to_R^* \varepsilon$, i.e.whether $C \notin Loh$:

1. compute D such that $val(D) = \rho(val(C))$.
2. check whether $val(D) \to_S^* \varepsilon$. If not, then accept, otherwise continue.
3. guess a position i such that the i-th symbol in $val(D)$ is '$\langle$' — this is the nondeterministic step.

4. compute $j = match(i)$.

5. check whether in $val(C)$, the i-th symbol is b and j-th symbol is d. If yes, then accept, otherwise reject.

It follows that the complementary problem, whether $C \in Loh$, is in **coNP**.

For the proof of **coNP**-hardness, Lohrey uses a polynomial reduction from the complement of the SUBSETSUM problem.

SUBSETSUM problem is the following question:

- Given a set S of positive integers $i_1, \ldots, i_n$ and a positive integer t, is there a subset $T \subseteq S$ such that the sum of elements of T equals to t?

SUBSETSUM is **NP**-complete (see e.g. [3]), therefore its complement is **coNP**-complete and therefore the reduction implies **coNP**-hardness of our problem.

We will not repeat here the reduction, but in Chapter 6, we will use the definition of a circuit encoding an instance of the SUBSETSUM problem.

Chapter 5

The CWC system

5.1 Basic description

We use the monoid $\mathcal{M} = M(A, R)$, where

$$\begin{aligned} A &= \{a, b, c, d\} \\ R &= \{(ac \to \varepsilon), (ad \to \varepsilon), (bc \to \varepsilon)\} \end{aligned}$$

introduced in the last chapter, and the **coNP**-complete language *Loh* from Definition 4.11, a special case of the compressed word problem for $\mathcal{M}$.

We are going to construct a proof system for the compressed word problem of $\mathcal{M}$, i.e. we define a system for proving that two circuits C, D, as compressed words from A^*, represent the same element of $\mathcal{M}$. We call this system CWC (for *Compressed Word Congruence*). In particular, CWC will be a proof system for the language *Loh*.

Let us describe the CWC system informally first:
Assume C, D are two circuits such that $val(C) \leftrightarrow_R^* val(D)$. Let $Ins(C)$ and $Ins(D)$ be the instruction sets, and $x_1, \ldots, x_n$ and $y_1, \ldots y_m$ be the variables of C and D, respectively. CWC starts with the elements of

$$R \cup Ins(C) \cup Ins(D)$$

as initial axioms, and derives congruences of the form

$$w_1 \to w_2$$

where $w_1, w_2 \in \{a, b, c, d, x_1, \ldots, x_n, y_1, \ldots, y_m\}^*$, using the following inference rules:

$$\overline{u \to u} \qquad \text{(a)}$$

$$\overline{\varepsilon u \to u} \qquad \text{(b)}$$

$$\overline{u\varepsilon \to u} \qquad \text{(c)}$$

$$\frac{}{(uv)w \to u(vw)} \qquad \text{(d)}$$

$$\frac{u \to v}{v \to u} \qquad \text{(e)}$$

$$\frac{u \to v \quad v \to w}{u \to w} \qquad \text{(f)}$$

$$\frac{u \to v \quad w \to x}{uw \to vx} \qquad \text{(g)}$$

The objective is to derive

$$x_n \to y_m$$

where x_n is the last variable in C and y_m the last variable in D.

Although the CWC system is designed to operate on compressed words, it works also for uncompressed words. In the case of uncompressed words $w_1, w_2 \in A^*$, the objective is to derive directly the congruence $w_1 \to w_2$. The uncompressed case is not that interesting, but we need the system to work also on uncompressed words, so that we can relate compressed words with their values.

Definition 5.1 *A* CWC proof *of a congruence X from axioms $Y_1, \ldots, Y_k$ is a sequence of congruences $Z_1, \ldots, Z_n$ such that*

$$Z_n = X$$

and for each Z_i, $i \in \{1, \ldots, n\}$ one of the following holds:

- *$Z_i = Y_j$, for some $j \in \{1, \ldots, k\}$*
- *Z_i is an instance of one of the rules (a)–(d)*
- *Z_i is derived from some Z_j, $j < i$, using the rule (e)*
- *Z_i is derived from some Z_j, Z_k, $j, k < i$, using one of the rules (f)–(g)*

The size *of a CWC-proof is the total number of symbols used in $Z_1, \ldots, Z_k$.*

Lemma 5.2 *Let C be a circuit with variables $x_1, \ldots, x_n$ and $val(C) = w$, where $w \in A^*$. Then*

$$x_n \to w$$

is provable from $Ins(C)$ in CWC.

Proof: By induction on i we will prove that

$$x_i \to w_i$$

where $w_i = val(C, x_i)$, is derivable.

For $i = 1$, the i-th instruction of C is of the form

$$x_1 \to w_1 \in A \cup \{\varepsilon\}$$

which is an axiom, so there is no derivation necessary.

For $i > 1$, the i-th instruction of C is either

$$x_i \to w_i \in A \cup \{\varepsilon\}$$

so again we get the result directly, or it is

$$x_i \to x_j x_k \quad \text{for } j, k < i$$

By induction assumption, $x_j \to w_j$ and $x_k \to w_k$ are derivable. Using rule (g) gives

$$x_j x_k \to w_j w_k$$

and using rule (f) together with $x_i \to x_j x_k$ gives

$$x_i \to w_j w_k$$

By the definition of the value of C at x_i, $w_j w_k = w_i$. □

Lemma 5.3 *Let $w_1, w_2 \in A^*$ be words such that $w_1 \leftrightarrow_R^* w_2$. Then*

$$w_1 \to w_2$$

is provable from R in CWC.

Proof: If $w_1 \leftrightarrow_R^* w_2$, then there is a chain $w_1 \leftrightarrow_R x_1 \leftrightarrow_R x_2 \leftrightarrow_R \ldots \leftrightarrow_R w_2$. Each step in the chain consists of one application of a rule in R. Without loss of generality, suppose the first step is of the form $w_1 \to_R x_1$ and the rule applied is $(ac \to \varepsilon)$. It means that $w_1 = uacv$ and $x_1 = uv$, where $u, v \in A^*$. To derive $uacv \to uv$, we proceed as follows:

1.	$v \to v$	rule (a)
2.	$ac \to \varepsilon$	rule from R
3.	$(ac)v \to \varepsilon v$	rule (g) on lines 1,2
4.	$\varepsilon v \to v$	rule (b)
5.	$(ac)v \to v$	rule (f) on lines 3,4
6.	$u \to u$	rule (a)
7.	$u((ac)v) \to uv$	rule (g) on lines 5,6

Every step in the chain can be derived similarly. □

Proposition 5.4 *Let C be a compressed word with variables $x_1, \ldots, x_n$, let D be a compressed word with variables $y_1, \ldots, y_m$. Assume that $val(C) \leftrightarrow_R^* val(D)$. Then*

$$x_n \to y_m$$

is provable from $R \cup Ins(C) \cup Ins(D)$ in CWC.

Proof: By 5.2, the relations $x_n \to val(C)$ and $y_m \to val(D)$ are provable. By 5.3, $val(C) \to val(D)$ is provable. Rule (e) gives $val(D) \to y_m$. Finally, double usage of rule (f) gives $x_n \to y_m$. □

Consider the set of all pairs of congruent words over A, both compressed and uncompressed. Let us call these congruent pairs *synonyms* and let us denote the set of all synonyms Syn. Note that Loh is a subset of Syn, consisting of those pairs which contain ε.

Proposition 5.5 *The CWC system is a proof system for Syn in the sense of Definition 3.1.*

Proof: Completeness of the system follows from Proposition 5.4. To prove its soundness, we have to show that whenever a relation $w_1 \to w_2$ is derivable, then the corresponding equivalence $\leftrightarrow_R^*$ holds. This follows from the fact that all relations in R and in the initial instruction sets of all circuits represent congruent words of M. Rules (a)-(g) preserve this congruence, therefore every relation derivable by these rules is an equivalence of $\leftrightarrow_R^*$.

To prove p-verifiability of the provability relation, we construct a DTM M, which gets as input two words C, D and a proof of $val(C) \leftrightarrow_R^* val(D)$, which has a size s. M works as follows: for each line in the proof, check whether it is an element of $Ins(C), Ins(D)$ or R, whether it is an application of one of the rules (a)–(d), or whether it is derived from earlier lines by one of the rules (e)–(g). This needs at most $\mathcal{O}(s^2)$ steps. □

Just to simplify the notation of proofs we introduce two more rules — substitution by zero and general associativity:

$$\frac{}{(uv)(wx) \to uvwx} \qquad \text{(as)}$$

(The right-hand side may contain also a different bracketing.)

$$\frac{u \to vxw \quad x \to \varepsilon}{u \to vw} \qquad \text{(sz)}$$

Both rules are derivable from rules (a)–(g), and therefore sound. Rule (as) can be derived as follows:

1.	$(uv)w \to u(vw)$	rule (d)
2.	$x \to x$	rule (a)
3.	$((uv)w)x \to (u(vw))x$	rule (g) on lines 1,2
4.	$(uv)(wx) \to ((uv)w)x$	rule (d)
5.	$(uv)(wx) \to (u(vw))x$	rule (f) on lines 4,3

By repeated usage of rules (d) and (g) we can derive also congruence of $u((vw)x)$ and $u(v(wx))$, therefore we can ignore the bracketing and write just $uvwx$ instead. But sometimes we will use the brackets even so, to emphasize the subword we work with.

Rule (sz) can be derived analogously to the proof of Lemma 5.3.

The CWC system allows us to operate directly with compressed words, without the need to "decompress" them first. If we want to check the congruence of two words in $\mathcal{M}$ (and, in particular, if we want to know whether a word can be reduced to ε), we do not have to expand the words to their full letter form. In some cases, this is a significant saving. A simple example follows:

Example 5.6 *Let C be a circuit in $2n+1$ variables $\{x_1, x_2, \ldots, x_{2n}, x_{2n+1}\}$.*

$x_1 \to a$
$x_i \to x_{i-1}x_{i-1}$ *for* $1 < i \leq n$
$x_{n+1} \to c$
$x_i \to x_{i-1}x_{i-1}$ *for* $n+1 < i \leq 2n$
$x_{2n+1} \to x_n x_{2n}$

Clearly, the value of C is:

$$val(C, x_n) = \underbrace{aa\ldots a}_{2^n}$$

$$val(C, x_{2n}) = \underbrace{cc\ldots c}_{2^n}$$

$$val(C) = val(C, x_{2n+1}) = \underbrace{aa\ldots a}_{2^n}\underbrace{cc\ldots c}_{2^n}$$

Using CWC, we can prove that $x_{2n+1} \to \varepsilon$ in polynomial time with respect to n. By induction on $k = 1, \ldots, n$ we prove that

$$x_k x_{n+k} \to \varepsilon$$

For $k = 1$ we have:

1.	$x_1 \to a$	*from $Ins(C)$*
2.	$x_k \to c$	*from $Ins(C)$*
3.	$x_1 x_k \to ac$	*rule (g) on lines 1,2*
4.	$ac \to \varepsilon$	*rule of R*
5.	$x_1 x_k \to \varepsilon$	*rule (f) on lines 3,4*

For $k+1 > 1$ we proceed:

1.	$x_{k+1} \to x_k x_k$	*from $Ins(C)$*
2.	$x_{n+k+1} \to x_{n+k}x_{n+k}$	*from $Ins(C)$*
3.	$x_{k+1}x_{n+k+1} \to x_k(x_k x_{n+k})x_{n+k}$	*rule (g) on lines 1,2 + (as)*
4.	$x_k x_{n+k} \to \varepsilon$	*induction assumption*
5.	$x_{k+1}x_{n+k+1} \to x_k x_{n+k}$	*rule (sz) on lines 3,4*
6.	$x_{k+1}x_{n+k+1} \to \varepsilon$	*rule (f) on lines 5,4*

The derivation uses at most $6n$ steps, each of length at most 7, so the size of the proof is in $\mathcal{O}(n)$.

5.2 Extended CWC

When we want to prove the congruence of two circuits C, D in CWC, we operate only with words constructed directly (by concatenation) from A and from subcircuits of C and D represented by $x_1, \ldots, x_n, y_1, \ldots, y_n$, respectively. This appears as an unnecessary limitation. In this chapter we extend the CWC system with a possibility to define arbitrary circuits.

The new, extended CWC system, denoted E-CWC, has one additional rule allowing, at any place in the proof, to introduce a new line:

$$\frac{}{z \to w} \qquad \text{(ext)}$$

if z is a variable that does not occur in the sets $Ins(C)$, $Ins(D)$ and nowhere in the proof so far, and w is an arbitrary word built from $a, b, c, d, x_1, \ldots, x_n$, $y_1, \ldots, y_m$ or variables introduced by (ext) earlier.

Proposition 5.7 *E-CWC is a proof system (in the sense of Definition 3.1) for the language Syn.*

Proof: Everything that is provable in CWC is also provable in E-CWC, and CWC is complete, hence E-CWC is complete.

The extension rule is sound, because for every new axiom $z \to w$ introduced in the proof, z is a variable that did not occur before and all its later occurences can be replaced by the sequence w. If all new variables introduced by using the (ext) rule are replaced in this way, we obtain a CWC proof, which is correct, because CWC is sound. Therefore, E-CWC is also sound.

The p-verifiability condition is also preserved. The verifying algorithm works in the same way as for CWC, checking just one more condition for each line in the proof — whether the lefthand side of the congruence is a new variable which did not occur before and whether the variable is not present in the righthand side. The time complexity of the algorithm is still in $\mathcal{O}(n^2)$. □

Chapter 6

Hard instances for CWC and E-CWC

Systems CWC and E-CWC seem to be analogous to propositional proof systems called Frege (F) and Extended Frege (EF) defined by Cook and Reckhow in [2]. Frege system operates with the formula to be proved and with other formulas it explicitly defines. Similarly CWC operates with words explicitly constructed from (variables representing) subcircuits of the initial circuit.

Extended Frege system has the power to abbreviate large formulas by introducing new variables for subformulas and, in effect, to operate with arbitrary circuits. This again is mirrored by the E-CWC.

There are no formulas known to be hard for F or EF, and it seems it will be analogously difficult to find some hard examples for CWC and E-CWC.

But since they are proof systems for a **coNP**-complete language (Loh, in our case, which is a subset of Syn), we expect that there exist hard instances, i.e. such that do not have a polynomially bounded proof. For these hard instances we shall look among those words which come from the reduction of the complementary problem to SUBSETSUM (see Chapter 4).

In the definition of a circuit (4.6) we allowed only concatenations of two symbols. Here, in order to make the circuit notation more transparent, we will write several concatenations at once. This will not affect the complexity, a step of the form

$$x_1 \to l_1 l_2 \dots l_k$$

can be performed using "pair-wise" concatenation, within at most $k-1$ steps and using at most $k-1$ new variable symbols.

The symbol l^n will denote the letter l concatenated n times.

The initial problem is:

- $i_1, \dots i_n$ and t are positive integers. Is it true that no subset of $\{i_1, \dots i_n\}$ adds up to t?

We denote $s_k = i_1 + \ldots + i_k$ and $s = s_n = i_1 + \ldots + i_n$.

An instance I of the problem consists of the integers $i_1, \ldots, i_n$ and t. We construct a circuit C_I corresponding to the instance I as follows:

$$\begin{array}{lll} & x_1 \to ba^{s+i_1}b & \\ \text{(first half)} & x_{k+1} \to x_k\, a^{s-s_k+i_{k+1}}\, x_k & \text{for } k = 1, \ldots, n-1 \\ & & \\ & x_{n+1} \to c^{s-t}dc^t & \\ \text{(second half)} & x_{k+1} \to x_k x_k & \text{for } k = n+1, \ldots, 2n \\ & & \\ \text{(full word)} & x_{2n+2} \to x_n x_{2n+1} & \end{array}$$

If we expand the first half of the circuit, $val(C_I, x_n)$, we obtain

$$ba^{s+i_1}ba^{s-i_1+i_2}ba^{s+i_1}ba^{s-i_1-i_2+i_3}ba^{s+i_1}ba^{s-i_1+i_2}ba^{s+i_1}b\ldots$$

We can divide this word into chunks of size $s+1$

$$ba^s \mid a^{i_1}ba^{s-i_1} \mid a^{i_2}ba^{s-i_2} \mid a^{i_2+i_1}ba^{s-i_2-i_1} \mid a^{i_3}ba^{s-i_3}$$

$$\mid a^{i_3+i_1}ba^{s-i_3+i_1} \mid a^{i_3+i_2}ba^{s-i_3+i_2} \mid a^{i_3+i_2+i_1}b \ \ldots$$

There are 2^n of these chunks, each contains exactly one letter b at a position corresponding to the sum of a subset of the integers i_k. There are n integers, 2^n possible subsets and each of the chunks represents one of the subsets.

The second half expanded ($val(C_I, x_{2n+1})$) also consists of 2^n chunks of size $s+1$, all of the same form

$$c^{s-t}dc^t$$

with exactly one letter d at the position of the required sum t (backwards).

When reducing the full word $val(C_I)$ by the rules of R, starting from the middle of the word, the only situation when we do not get to the empty word ε is, when a pair of the letters b and d meet. This can happen only if there is a subset of i_k's with the sum equal to t. In this way, we reduced the complement of SUBSETSUM to the language *Loh* (for full details see [6]).

As a consequence of the reduction and Proposition 3.3 (Cook-Reckhow) we get the following result:

Proposition 6.1 *The circuits $C_I \in Loh$ defined above cannot have polynomially bounded proofs in either CWC or E-CWC, unless* $\mathbf{NP} = \mathbf{coNP}$.

Bibliography

[1] Buss S., Impagliazzo R., Krajíček J., Pudlák P., Razborov A.A., Sgall J. (1997): *Proof complexity in algebraic systems and bounded depth Frege systems with modular counting*. Computational Complexity **6**, 256–298.

[2] Cook S.A., Reckhow R.A. (1979): *The relative efficiency of propositional proof systems*. The Journal of Symbolic Logic **44**, 36–50.

[3] Garey M.R., Johnson D.S. (1979): *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, New York.

[4] Holub Š: *Složitost pro kryptografii*. Unpublished lecture notes, available at http://www.karlin.mff.cuni.cz/∼holub/skripta/slozitost.ps

[5] Krajíček J. (2005): *Proof complexity*. In: Proc. 4th European congress of mathematics (ed. A. Laptev), EMS, Zurich 221–231.

[6] Lohrey M. (2006): *Word problems and membership problems on compressed words*. SIAM Journal on Computing **35**, 1210–1240.

[7] Papadimitriou C.H. (1995): *Computational Complexity*. Addison-Wesley Longman, Reading.